下载APP

22 | 支付订单信息：如何高效解决for循环产生的内存溢出?

2021-05-10 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼

时长 20:03 大小 18.38M

你好，我是高楼。

今天，我们来优化支付订单接口。通过这个接口，我们来看看怎么高效解决 for 循环产生的内存溢出问题。

对于 JVM 内存溢出或泄露来说，通常性能人员都能定位到一个应用 hang 住了。但是，要想进一步判断出应用 hang 住的原因，并没有那么容易做到。因为内存大时做堆 Dump 比较费时，更重要的一点是，要想把堆里面的对象和栈关联起来是需要足够的经验和时间的。这也是其中的难点之一。

这节课我就带你来看看怎么解决这个难点。

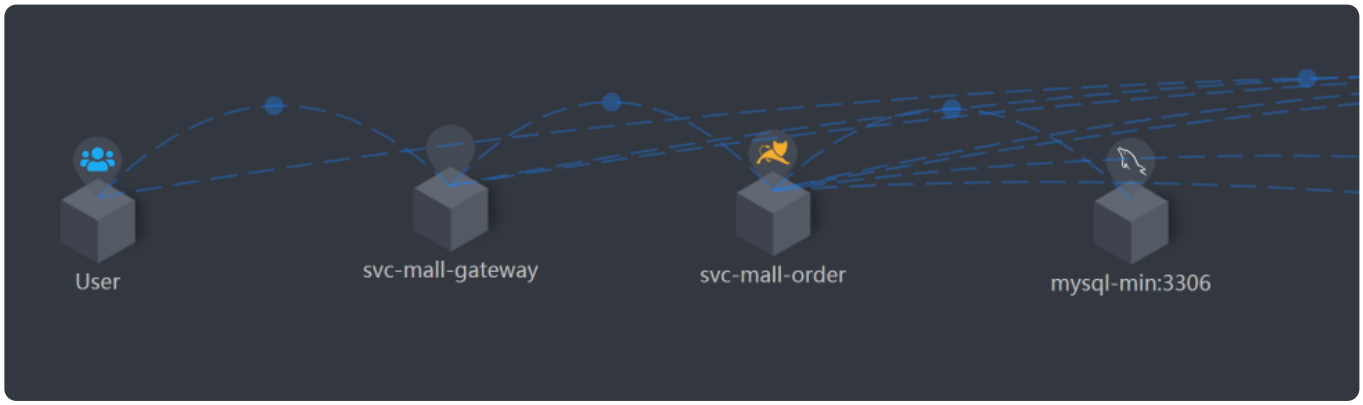
不过，在此之前，我们会先处理一个熟悉的问题，就是数据库表加索引。很显然，我们在测试这个接口时又遇到它了。虽然我在第 16 讲中给你重点讲过这个问题，但是这一次的每秒全表扫描比之前要高得多。通过这次的讲解，我希望你能明白只要存在全表扫描，CPU 消耗很快就会达到 100%。同时，也希望你能借此看清楚全表扫描对 CPU 消耗的影响。

场景运行数据

首先，我们来运行一下场景：

这是一个典型的 TPS 太低、响应时间不断上升的性能瓶颈，对于这种瓶颈的分析逻辑，我在前面的课程里已经写过很多次了，相信你已经掌握。下面我们来看一下具体的问题是什么。

架构图



这个接口的链路比较简单：User - Gateway - Order - MySQL，我们大概记在脑子里就好。

第一阶段

在这里我就不拆分时间了，我们直接来看全局监控。因为第一阶段的问题相对来说比较简单，只是性能瓶颈的表现形式和之前不太一样。

全局监控分析

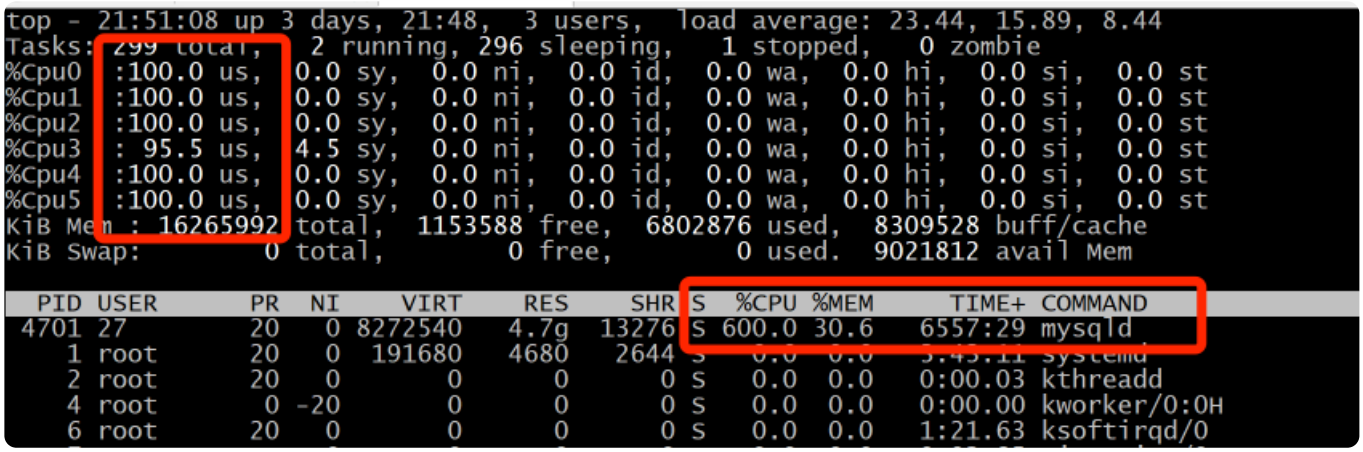
全局监控的数据如下：

看到这张图，你是不是有一种终于见到典型性能瓶颈的感觉？CPU 使用率这么高！那还不简单？打栈看代码呀！

不过，我们得先查一下是什么东西导致 k8s-worker-1 的 CPU 使用率这么高的。这个 worker 上运行的服务如下：

```
[root@k8s-master-2 ~]# kubectl get pods -o wide | grep worker-1
mysql-min-6685c9ff76-4m5xr                  1/1     Running     0          4d2
skywalking-es-init-ls7j5                    0/1     Completed   0          4d1
[root@k8s-master-2 ~]#
```

可以看到，有两个服务在这个 worker 上跑着，一个是初始化容器，另一个是 MySQL。初始化容器已经是完成的状态，那 CPU 使用率高肯定是因为 MySQL 了。因此，我们就进到容器中，执行下 top，看看资源消耗到什么程度。


top - 21:51:08 up 3 days, 21:48, 3 users, load average: 23.44, 15.89, 8.44
Tasks: 299 total, 2 running, 296 sleeping, 1 stopped, 0 zombie
%Cpu0 :100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 :100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu2 :100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 : 95.5 us, 4.5 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu4 :100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu5 :100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 16265992 total, 1153588 free, 6802876 used, 8309528 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 9021812 avail Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
4701 27 20 0 8272540 4.7g 13276 S 600.0 30.6 6557:29 mysqld
1 root 20 0 191680 4680 2644 S 0.0 0.0 3:45.11 systemd
2 root 20 0 0 0 0 S 0.0 0.0 0:00.03 kthreadd
4 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 kworker/0:0H
6 root 20 0 0 0 0 S 0.0 0.0 1:21.63 ksoftirqd/0

什么情况？CPU 使用率这么高吗？既然这样，我们就得来查一下 MySQL 的全局监控了。

在这里，**查看全局监控其实是分析 MySQL 的必要步骤**。如果直接查看 trx 表，中间其实是有些跳跃的，因为查看 MySQL 的全局监控数据，才是承上启下的一步。我现在把这个过程写全一些，以免你产生困惑。

于是，我们执行一个 mysqlreport 命令，看看 mysql 的全局监控数据是怎样的。这里我截取了其中的一些重要信息：

```
1  __ Questions ___________________________________________________________
2  Total          307.93M   912.0/s
3    +Unknown     201.91M   598.0/s  %Total:  65.57
4    DMS           43.20M   128.0/s           14.03
5    Com_          32.90M    97.5/s           10.69
6    QC Hits       29.91M    88.6/s            9.71
7    COM_QUIT         389     0.0/s            0.00
8  Slow 20 ms     273.66k     0.8/s            0.09  %DMS:   0.63  Log:
9  DMS            43.20M   128.0/s           14.03
10   SELECT       32.39M    95.9/s           10.52         74.98
11   INSERT       10.64M    31.5/s            3.46         24.63
12   UPDATE      170.15k     0.5/s            0.06          0.39
13   REPLACE           0       0/s            0.00          0.00
14   DELETE            0       0/s            0.00          0.00
15 Com_           32.90M    97.5/s           10.69
16   set_option   21.98M    65.1/s            7.14
17   commit       10.70M    31.7/s            3.48
18   admin_comma 137.68k     0.4/s            0.04
19
20
21 __ SELECT and Sort _____________________________________________________
22 Scan           20.04M    59.4/s %SELECT:  61.88
23 Range               0       0/s            0.00
24 Full join          32     0.0/s            0.00
```

```
Range check          0       0/s            0.00
Full rng join        0       0/s            0.00
Sort scan          120     0.0/s
Sort range       2.41k     0.0/s
Sort mrg pass        0       0/s
```

你看，DMS 中的 select 占比比较大。其实，如果只是 select 的占比比较大的话，倒不是什么大事，关键是在下面的数据中还有一个 Scan（全表扫描），而全表扫描是典型的性能问题点。

看到这里，我想你应该非常清楚我接下来的套路了吧，就是找 SQL，看执行计划，然后确定优化方案。如果你不太清楚，可以再看一下第 16 讲或第 20 讲，其中都有描述。

定向监控分析

于是，我们现在进入到了定向监控分析阶段。通过查看 innodb_trx 表，我们看到了 SQL 中执行慢的语句，它的执行计划如下：

```
EXPLAIN SELECT
  o.id,
  o.order_sn,
  o.coupon_id,
  o.integration,
  o.member_id,
  ot.id ot_id,
  ot.product_name ot_product_name,
  ot.product_sku_id ot_product_sku_id,
  ot.product_sku_code ot_product_sku_code,
  ot.product_quantity ot_product_quantity
FROM
  oms_order o
  LEFT JOIN oms_order_item ot
    ON o.id = ot.order_id
WHERE o.id = 57446 ;
```

1 结果 2 信息 3 表数据 4 信息

（只读）

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	o	(NULL)	const	PRIMARY	PRIMARY	8	const	1	100.00	(NULL)
1	SIMPLE	ot	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	170561	100.00	Using where

这是很典型的全表扫描，虽然数据量并不大，但是我们也要添加索引。添加索引的语句如下：

```
mysql> ALTER TABLE `oms_order_item` ADD INDEX index_name ( `order_id` );
```

这里你要注意一下，在创建索引的时候，如果数据量太大，创建索引可能会卡住很长时间，这要取决于机器单 CPU 的能力。

优化效果

添加索引之后，我们直接来看一下优化效果：

你看，TPS 要上千了！

其实，对于 SQL 的优化，如果我们只是加一个索引，那就算是非常简单的步骤了，并且效果也会非常好，TPS 增加上千倍、甚至上万倍都有可能。可是，如果优化涉及到了业务逻辑，那就麻烦一些了。

如果你觉得这节课只是为了给你讲一个加索引的案例，那你就有些单纯了。下面，我们来看一个复杂的问题。

第二阶段

在我接着执行压力的时候，看到了一个非常有意思的情况，我们来一起折腾折腾！

场景运行数据

你看这张图，在压力持续大概 6 分钟之后，TPS 不稳定也就算了，居然还掉下来了！你掉下来也就算了，居然还断开了！你断开了也就算了，响应时间居然也不增加！

这可怎么分析呢？想要拆分时间都没有一个适合的理由呀！

这时候，就得用上哥的性能分析决策树了。我们把相关的全局监控计数器都看一看，一层层查下去，还好这个接口的逻辑链路也不怎么长。

全局监控分析

按照高老师的习惯，我们首先来看全局监控：

服务器资源总览表【JOB：node-exporter，主机总数：12】

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	2.53 week	7.64 GiB	4	0.43	9.44%	24.23%	29.95%	0 B/s	69.10 kB/s	290	43	590.65 kbps	418.89 kbps
k8s-master-2	k8s-master-2	5.11 week	7.64 GiB	4	0.63	16.13%	23.51%	29.95%	0 B/s	66.45 kB/s	290	43	475.82 kbps	475.82 kbps
k8s-master-3	k8s-master-3	5.12 week	7.64 GiB	4	0.20	7.66%	24.95%	29.95%	0 B/s	70.11 kB/s	318	132	833.86 kbps	1.05 Mbps
k8s-worker-1	k8s-worker-1	4.66 day	15.51 GiB	6	0.19	3.93%	44.83%	58.14%	9.21 kB/s	935.42 kB/s	35	112	528.96 kbps	553.02 kbps
k8s-worker-2	k8s-worker-2	4.66 day	15.51 GiB	6	0.09	2.42%	14.85%	29.95%	9.21 kB/s	2.13 kB/s	28	77	36.12 kbps	33.21 kbps
k8s-worker-3	k8s-worker-3	1.59 week	15.51 GiB	6	1.35	25.35%	59.27%	34.34%	19.43 kB/s	18.47 kB/s	33	264	790.27 kbps	913.93 kbps
k8s-worker-4	k8s-worker-4	4.03 week	15.51 GiB	6	1.15	9.86%	34.77%	35.88%	9.21 kB/s	24.92 kB/s	56	2 K	1.16 Mbps	1.45 Mbps
k8s-worker-5	k8s-worker-5	4.02 week	15.51 GiB	6	0.40	14.79%	22.53%	36.42%	9.21 kB/s	4.17 kB/s	28	123	804.56 kbps	766.88 kbps
k8s-worker-6	k8s-worker-6	4.02 week	15.51 GiB	6	0.44	14.50%	21.04%	39.29%	9.21 kB/s	3.11 kB/s	29	108	811.98 kbps	778.89 kbps
k8s-worker-7	k8s-worker-7	3.72 week	15.51 GiB	4	0.11	7.92%	28.87%	51.48%	9.21 kB/s	9.23 kB/s	28	125	983.65 kbps	948.87 kbps
k8s-worker-8	k8s-worker-8	2.55 week	15.51 GiB	4	0.43	15.59%	49.50%	53.13%	0 B/s	10.51 kB/s	27	316	2.30 Mbps	2.35 Mbps
k8s-worker-9	k8s-worker-9	3.72 week	15.51 GiB	6	1.66	19.34%	25.80%	29.95%	9.21 kB/s	5.82 kB/s	29	80	6.47 Mbps	14.73 Mbps

从这张图上，我们什么也没看出来。所以，我们接着查性能分析决策树，一层层地看下去。当看到 Order 的 GC 健康状态时，我们看到了下面这些数据：

复制代码

```
[root@svc-mall-order-568bd9b79-twhcw /]# jstat -gcutil 1 1s
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
100.00   0.00 100.00 100.00  95.06  92.82   1182   34.966   495 3279.704 3314.
100.00   0.00 100.00 100.00  95.06  92.82   1182   34.966   495 3279.704 3314.
```

```
100.00   0.00 100.00 100.00  95.06  92.82   1182   34.966   495 3279.704 3314.
100.00   0.00 100.00 100.00  95.06  92.82   1182   34.966   495 3279.704 3314.
 90.88   0.00 100.00 100.00  95.08  92.82   1182   34.966   495 3286.621 3321.
100.00   0.00 100.00 100.00  95.08  92.82   1182   34.966   496 3286.621 3321.
100.00   0.00 100.00 100.00  95.08  92.82   1182   34.966   496 3286.621 3321.
100.00   0.00 100.00 100.00  95.08  92.82   1182   34.966   496 3286.621 3321.
100.00   0.00 100.00 100.00  95.08  92.82   1182   34.966   496 3286.621 3321.
100.00   0.00 100.00 100.00  95.08  92.82   1182   34.966   496 3286.621 3321.
100.00   0.00 100.00 100.00  95.08  92.82   1182   34.966   496 3286.621 3321.
.........................
```

有没有感到绝望？内存泄漏了！年轻代和年老代的内存都用到 100% 了！即便是 FullGC 之后，内存也没回收下来，可怜呀。

定向监控分析

既然是内存被用完了，那我们自然要查一下是什么样的对象把内存用完了。所以，我们进到容器里面，执行 jmap -histo 1|more 看一眼：

复制代码

```
num     #instances         #bytes  class name
----------------------------------------------
   1:      49727866     1397691896  [
   2:      12426103      795269200  [[
   3:      12426038      397633216  com.mysql.cj.protocol.a.result.ByteArrayRo
   4:       2002596      384498432  com.dunshan.mall.order.domain.OmsOrderDeta
   5:      12426082      198817312  com.mysql.cj.protocol.a.MysqlTextValueDeco
   6:       2070085      182840264  [Ljava.lang.Object
   7:       6008660      144207840  java.lang.Lon
   8:       2207452      132116320  [
   9:       4072895       97749480  java.util.ArrayLis
  10:       2002690       80107600  org.apache.ibatis.cache.CacheKe
  11:       2039613       65267616  java.util.HashMap$Nod
  12:       2197616       52742784  java.lang.Strin
  13:         14736       23246672  [Ljava.util.HashMap$Node
  14:         36862        3243856  java.lang.reflect.Metho
  15:         97195        3110240  java.util.concurrent.ConcurrentHashMap$Nod
  16:         62224        2986752  java.util.HashMa
  17:         19238        2452264  [
  18:         21482        2360328  java.lang.Clas
  19:         26958        1078320  java.util.LinkedHashMap$Entr
.........................
```

从中我们似乎找到了问题点。你看，这里面有一个 MySQL 的 result 占的内存还挺大，同时，在它的下面我们也看到了 OmsOrderDetail 类，这个类是用来在数据库中查询订单的详细信息的。

从逻辑上来讲，我们看订单的详细信息，实际上是想查询数据库中的信息，进而把查询出来的数据放到应用的内存中。所以，MySQL 的 result 查的数据越多，就会导致应用的 JVM 内存消耗越大。

你也许会想，接下来是不是直接去看 OmsOrderDetail 的代码就可以了？你可以去看，但是，我们这个案例并没有那么直接。因为我们已经知道代码了，逻辑也梳理清楚了，所以，再去查看代码其实也看不出什么问题来。

那为什么 JVM 内存消耗会高呢？这里我们就要查一下线程在做什么动作了：

复制代码

```
-- top
[root@k8s-worker-3 ~]# docker exec -it 66d3639cf4a8 /bin/bash
[root@svc-mall-order-568bd9b79-twhcw /]# top
top - 16:10:50 up 11 days,  2:37,  0 users,  load average: 3.92, 4.77, 3.35
Tasks:   4 total,   1 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu0  : 46.7 us,  8.6 sy,  0.0 ni, 43.6 id,  0.0 wa,  0.0 hi,  0.7 si,  0.3 s
%Cpu1  : 23.3 us,  9.2 sy,  0.0 ni, 66.1 id,  0.0 wa,  0.0 hi,  1.0 si,  0.3 s
%Cpu2  : 50.0 us,  7.2 sy,  0.3 ni, 41.4 id,  0.0 wa,  0.0 hi,  0.7 si,  0.3 s
%Cpu3  : 46.4 us,  8.5 sy,  0.0 ni, 43.7 id,  0.0 wa,  0.0 hi,  1.0 si,  0.3 s
%Cpu4  : 50.5 us,  8.0 sy,  0.0 ni, 41.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu5  : 50.2 us,  3.1 sy,  0.0 ni, 46.1 id,  0.0 wa,  0.0 hi,  0.7 si,  0.0 s
KiB Mem : 16265992 total,   171760 free,  9077080 used,  7017152 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  6676508 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0 8788300   4.2g  13860 S 127.9 27.1 115:17.15 java
  575 root      20   0   11828   1776   1328 S   0.0  0.0   0:00.01 sh
  789 root      20   0   11964   1980   1484 S   0.0  0.0   0:00.02 bash
  802 root      20   0   56232   2012   1432 R   0.0  0.0   0:00.05 to


-- top -Hp 1
top - 16:11:39 up 11 days,  2:38,  0 users,  load average: 8.87, 6.09, 3.87
Threads:  85 total,   1 running,  84 sleeping,   0 stopped,   0 zombie
%Cpu0  : 55.6 us,  7.1 sy,  0.0 ni, 36.6 id,  0.0 wa,  0.0 hi,  0.3 si,  0.3 s
%Cpu1  : 41.3 us,  3.8 sy,  0.0 ni, 54.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu2  : 30.4 us,  9.9 sy,  0.0 ni, 59.4 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 s
%Cpu3  : 60.3 us,  6.7 sy,  0.0 ni, 32.7 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 s
```

```
%Cpu4  : 21.2 us,  9.2 sy,  0.0 ni, 69.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu5  : 45.6 us, 10.1 sy,  0.3 ni, 43.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 s
KiB Mem : 16265992 total,   197656 free,  9071444 used,  6996892 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  6681848 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    7 root      20   0 8788300   4.2g  13836 R 96.0 27.1  70:13.42 VM Thread
   26 root      20   0 8788300   4.2g  13836 S  0.7 27.1   0:05.70 VM Periodic
```

执行了上面两个命令之后，你有没有注意到只有一个线程在消耗 CPU？根据我们前面查看的 GC 状态，这个线程应该在忙着做 FullGC。我们打印栈信息来确认一下，果然是它！

复制代码

```
"VM Thread" os_prio=0 tid=0x00007fb18c0f4800 nid=0x7 runnable
```

到这里，我们下一步该怎么做就非常清晰了，那就是打印一个堆 Dump，来看看对象在内存中的消耗比例。

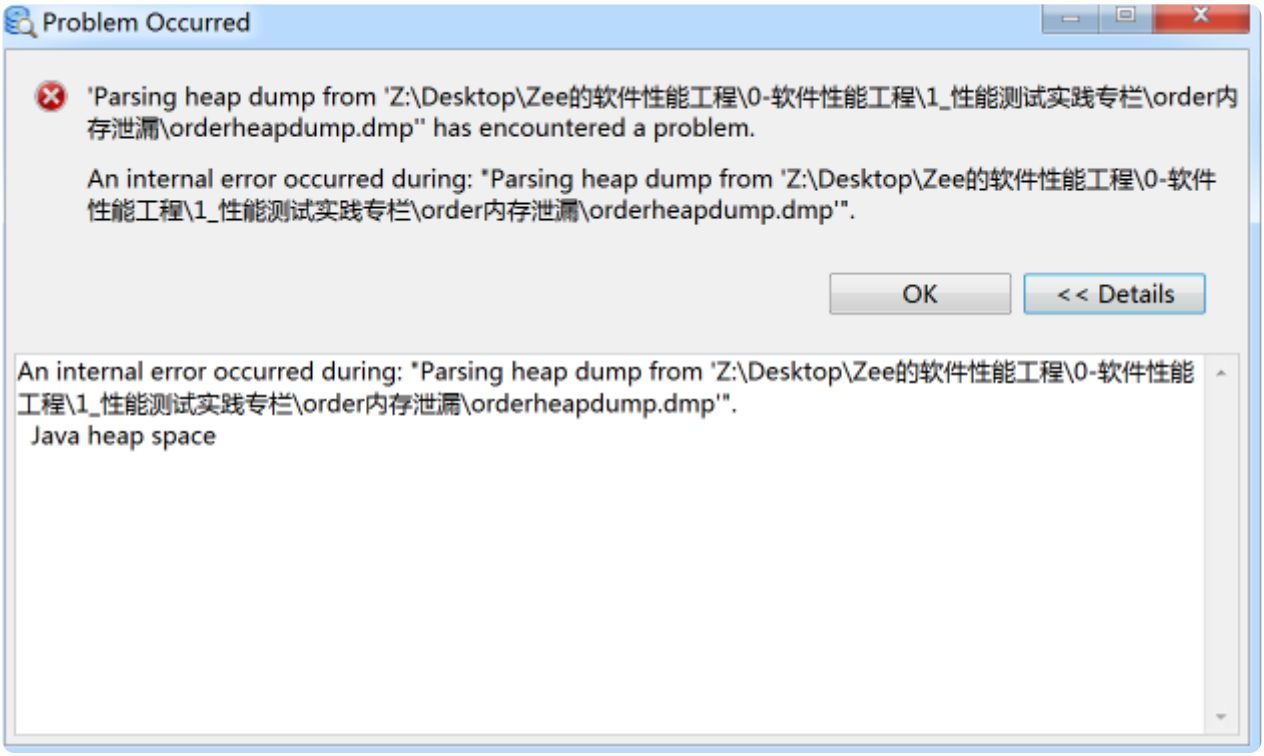
所以，我们现在执行下面这个命令，来生成堆 Dump。

复制代码

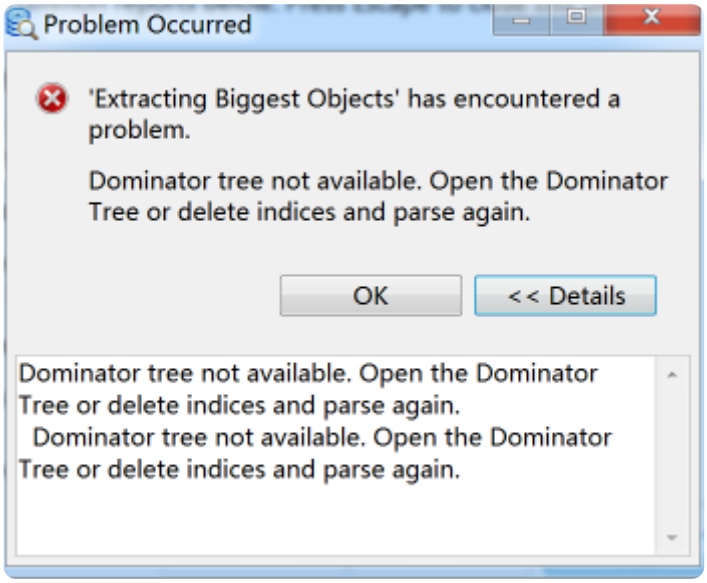
```
jmap -dump:format=b,file=文件名 pid
```

然后，我们再把生成的堆 Dump 下载下来，用 MAT 打开。

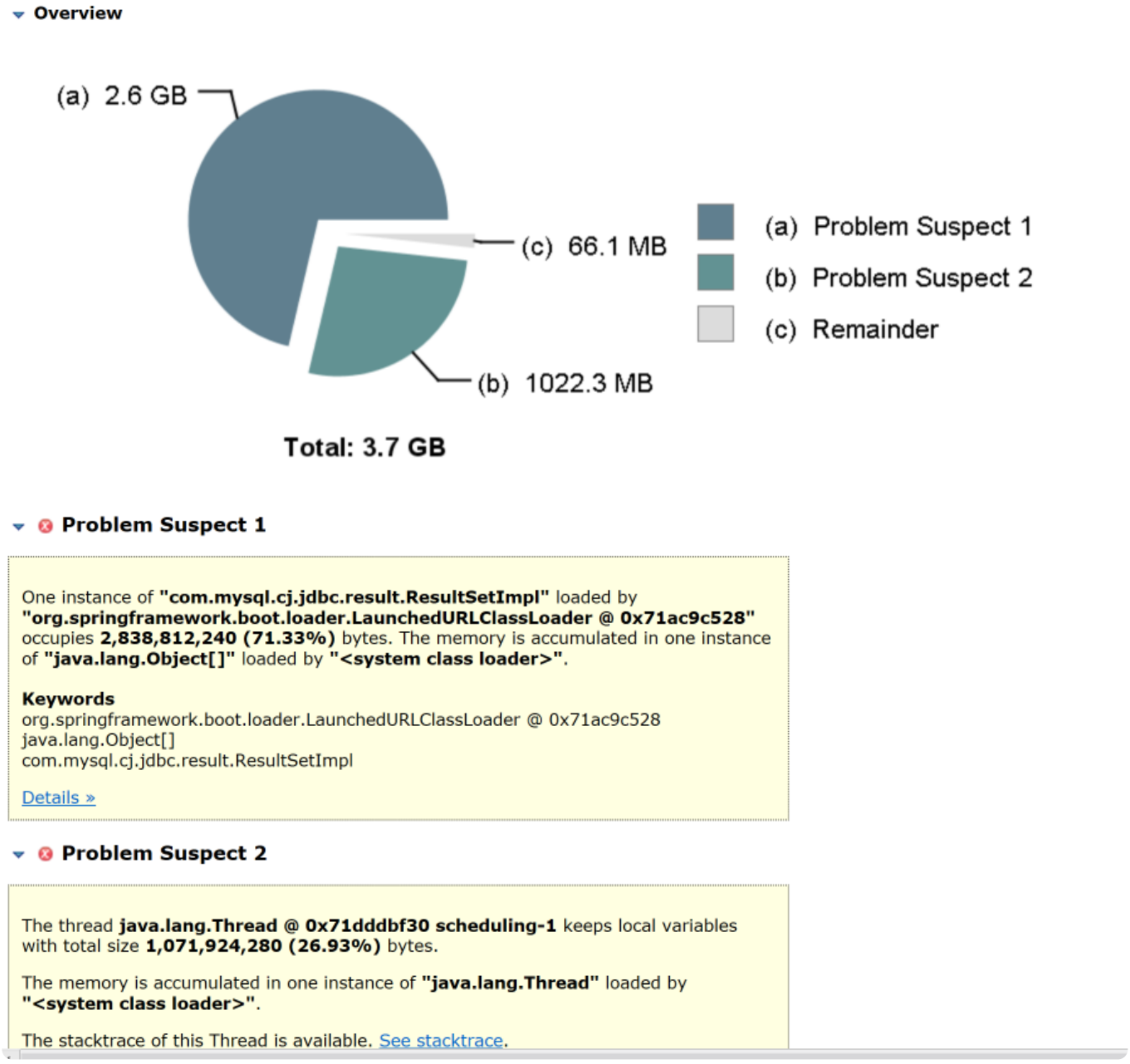
在我打开堆文件的过程中出现了一个小插曲，这也是你需要留意的地方，那就是如果堆 Dump 的内存太大的话，我们打开堆 Dump 就会报这样的错：



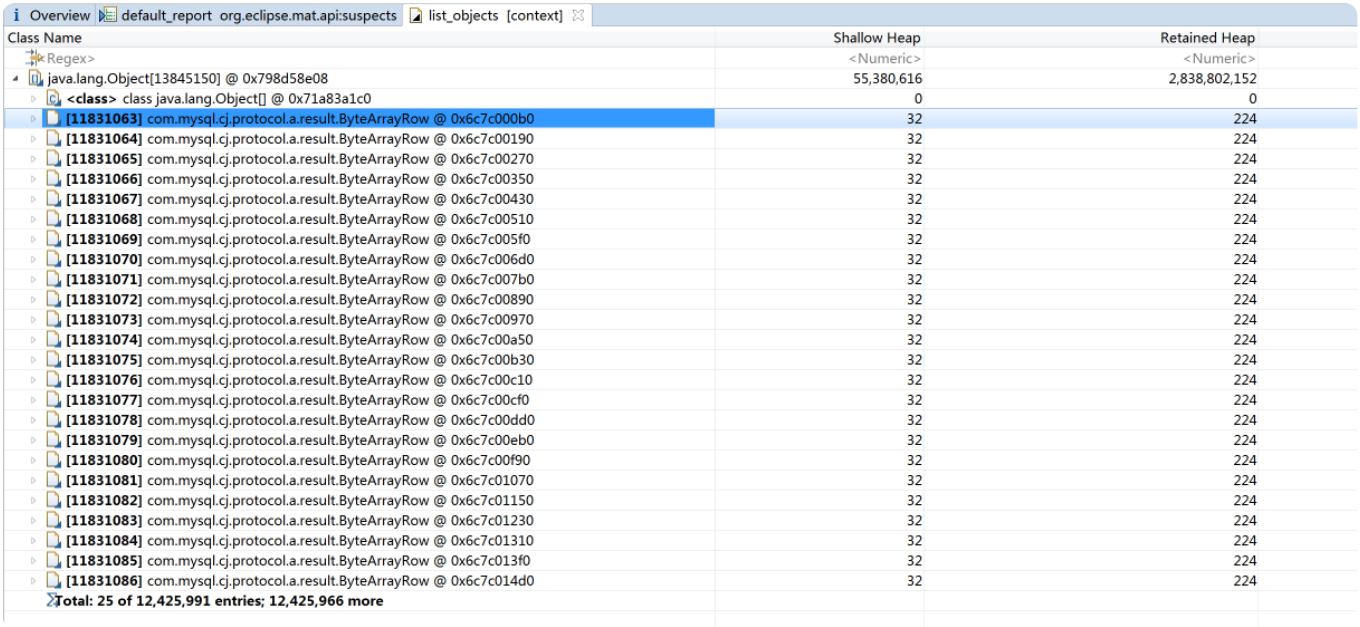
这个时候我们就要到 MemoryAnalyzer.ini 文件中，把 JVM 的最大值参数 -Xmx 给调大。-Xmx 的默认值是 1G，至于要调到多大就要看你想打开多大的文件了。如果 -Xmx 调整得不够大，还会出现下面这样的错误：



在我们费了九牛二虎之力，终于打开堆文件后，看到这样的信息：

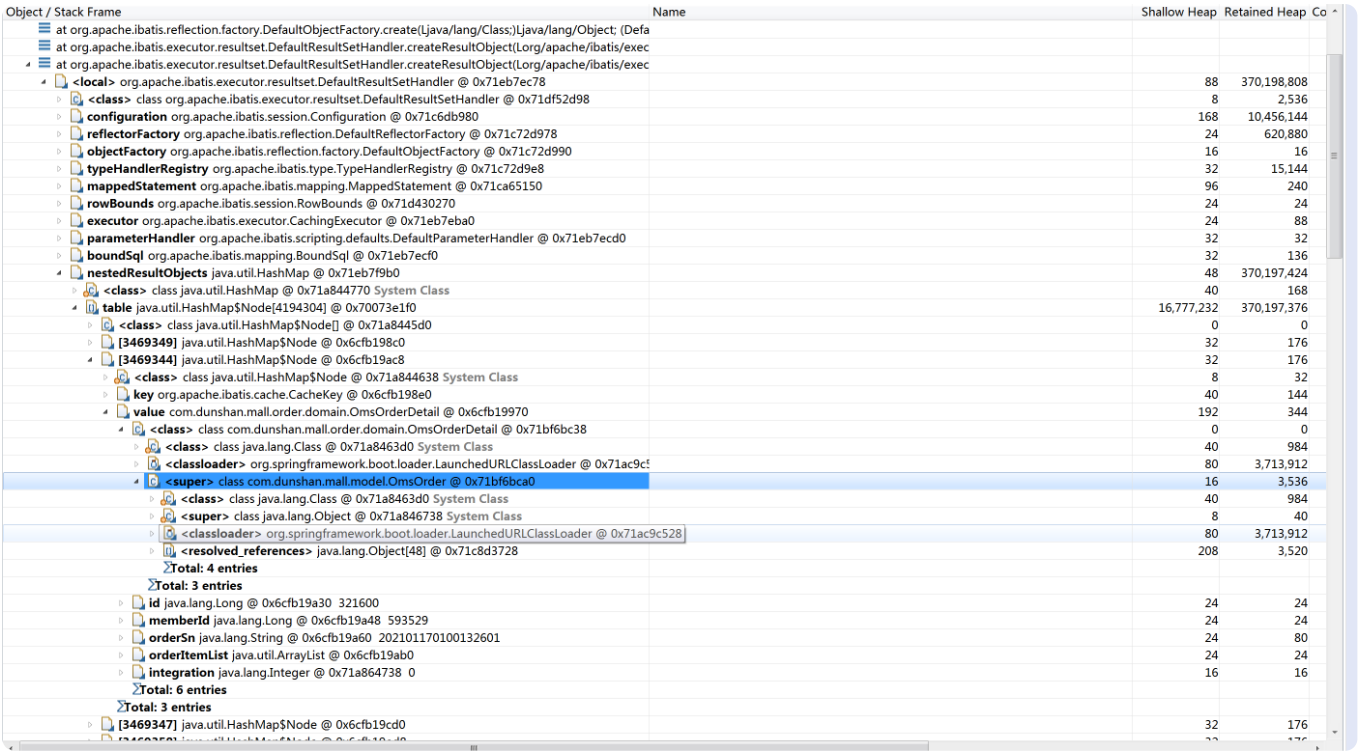


有一段内存居然消耗了 2.6G！这个可疑的内存点在上图中也出现了（用 MAT 或 jmap 来看对象的内存消耗，是两种不同的手段，你可以自己选择。只是在 MAT 上，我们可以看到可疑的内存消耗点的提醒，而在 jmap 中是不会给出这样的提醒的，需要我们自己判断），我们点进去看一眼，看看可疑的内存点里具体的对象是什么：



Class Name	Shallow Heap	Retained Heap
Regex>	<Numeric>	<Numeric>
java.lang.Object[13845150] @ 0x798d58e08	55,380,616	2,838,802,152
<class> class java.lang.Object[] @ 0x71a83a1c0	0	0
[11831063] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c000b0	32	224
[11831064] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00190	32	224
[11831065] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00270	32	224
[11831066] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00350	32	224
[11831067] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00430	32	224
[11831068] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00510	32	224
[11831069] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c005f0	32	224
[11831070] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c006d0	32	224
[11831071] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c007b0	32	224
[11831072] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00890	32	224
[11831073] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00970	32	224
[11831074] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00a50	32	224
[11831075] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00b30	32	224
[11831076] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00c10	32	224
[11831077] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00cf0	32	224
[11831078] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00dd0	32	224
[11831079] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00eb0	32	224
[11831080] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c00f90	32	224
[11831081] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c01070	32	224
[11831082] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c01150	32	224
[11831083] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c01230	32	224
[11831084] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c01310	32	224
[11831085] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c013f0	32	224
[11831086] com.mysql.cj.protocol.a.result.ByteArrayRow @ 0x6c7c014d0	32	224
Total: 25 of 12,425,991 entries; 12,425,966 more		

你看，确实是 SQL 返回的数据量比较大，在上图的列表中居然有 1 千多万条记录。我们再把相应的栈展开看看：



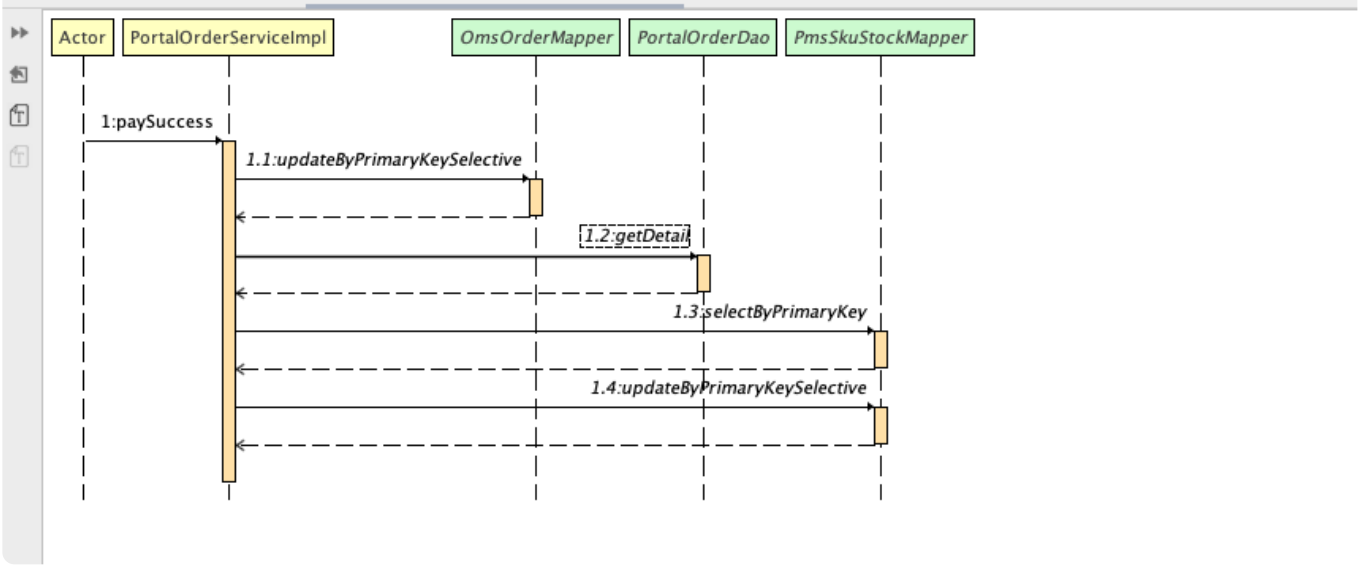
Object / Stack Frame	Name	Shallow Heap	Retained Heap
at org.apache.ibatis.reflection.factory.DefaultObjectFactory.create(Ljava/lang/Class;)Ljava/lang/Object; (Defa			
at org.apache.ibatis.executor.resultset.DefaultResultSetHandler.createResultObject(Lorg/apache/ibatis/exec			
at org.apache.ibatis.executor.resultset.DefaultResultSetHandler.createResultObject(Lorg/apache/ibatis/exec			
<local> org.apache.ibatis.executor.resultset.DefaultResultSetHandler @ 0x71eb7ec78		88	370,198,808
<class> class org.apache.ibatis.executor.resultset.DefaultResultSetHandler @ 0x71df52d98		8	2,536
configuration org.apache.ibatis.session.Configuration @ 0x71c6db980		168	10,456,144
reflectorFactory org.apache.ibatis.reflection.DefaultReflectorFactory @ 0x71c72d978		24	620,880
objectFactory org.apache.ibatis.reflection.factory.DefaultObjectFactory @ 0x71c72d990		16	16
typeHandlerRegistry org.apache.ibatis.type.TypeHandlerRegistry @ 0x71c72d9e8		32	15,144
mappedStatement org.apache.ibatis.mapping.MappedStatement @ 0x71ca65150		96	240
rowBounds org.apache.ibatis.session.RowBounds @ 0x71d430270		24	24
executor org.apache.ibatis.executor.CachingExecutor @ 0x71eb7eba0		24	88
parameterHandler org.apache.ibatis.scripting.defaults.DefaultParameterHandler @ 0x71eb7ecd0		32	32
boundSql org.apache.ibatis.mapping.BoundSql @ 0x71eb7ecf0		32	136
nestedResultObjects java.util.HashMap @ 0x71eb7f9b0		48	370,197,424
<class> class java.util.HashMap @ 0x71a844770 System Class		40	168
table java.util.HashMap$Node[4194304] @ 0x70073e1f0		16,777,232	370,197,376
<class> class java.util.HashMap$Node[] @ 0x71a8445d0		0	0
[3469349] java.util.HashMap$Node @ 0x6cfb198c0		32	176
[3469344] java.util.HashMap$Node @ 0x6cfb19ac8		32	176
<class> class java.util.HashMap$Node @ 0x71a844638 System Class		8	32
key org.apache.ibatis.cache.CacheKey @ 0x6cfb198e0		40	144
value com.dunshan.mall.order.domain.OmsOrderDetail @ 0x6cfb19970		192	344
<class> class com.dunshan.mall.order.domain.OmsOrderDetail @ 0x71bf6bc38		0	0
<class> class java.lang.Class @ 0x71a8463d0 System Class		40	984
<classloader> org.springframework.boot.loader.LaunchedURLClassLoader @ 0x71ac9c5		80	3,713,912
<super> class com.dunshan.mall.model.OmsOrder @ 0x71bf6bca0		16	3,536
<class> class java.lang.Class @ 0x71a8463d0 System Class		40	984
<super> class java.lang.Object @ 0x71a846738 System Class		8	40
<classloader> org.springframework.boot.loader.LaunchedURLClassLoader @ 0x71ac9c528		80	3,713,912
<resolved_references> java.lang.Object[48] @ 0x71c8d3728		208	3,520
Total: 4 entries			
Total: 3 entries			
id java.lang.Long @ 0x6cfb19a30 321600		24	24
memberId java.lang.Long @ 0x6cfb19a48 593529		24	24
orderSn java.lang.String @ 0x6cfb19a60 202101170100132601		24	80
orderItemList java.util.ArrayList @ 0x6cfb19ab0		24	24
integration java.lang.Integer @ 0x71a864738 0		16	16
Total: 6 entries			
Total: 3 entries			
[3469347] java.util.HashMap$Node @ 0x6cfb19cd0		32	176

看到 OmsOrderDetail 这个类了没？你可能会想，是支付订单信息这个接口有问题！但是证明还不足，我们要确定 OmsOrderDetail 是不是在这个接口中调用或生成的，才能判断出是不是这个接口的问题。

由于当前使用的接口是 paySuccess，我们看一下 paySuccess 的调用逻辑，发现 paySuccess 有一个 getDetail 函数。看到这个“getDetail”，还有其中这个“Detail”字

符，你是不是感觉和 OmsOrderDetail 可以对应上？那我们就来查看一下 getDetail 对应的代码，看看它和 OmsOrderDetail 之间是什么关系：

```
342
343         //撤消锁定库存，扣减真实库存
344         OmsOrderDetail orderDetail = portalOrderDao.getDetail(orderId);
345         List<OmsOrderItem> orderItemList = orderDetail.getOrderItemList();
346
347         //修改代码
348         int count = 1;
349         for (OmsOrderItem omsOrderItem : orderItemList) {
350             PmsSkuStock pmsSkuStock = skuStockMapper.selectByPrimaryKey(omsOrderItem.getProductSkuId());
351             //lock_stock
352             pmsSkuStock.setStock(pmsSkuStock.getStock() - omsOrderItem.getProductQuantity());
```



不难发现，getDetail 是一个 OmsOrderDetail 类。这么看来，我们的接口确实用到了 OmsOrderDetail 类，你是不是有一种抓住元凶的兴奋感？别着急，让人无奈的事情总是会出现的，下面我们看一下这段代码对应的 SQL 语句是一个什么样的逻辑。

通过查看代码，可以看到这个接口中有两个 update 和一个 select，这三个语句分别是：

复制代码

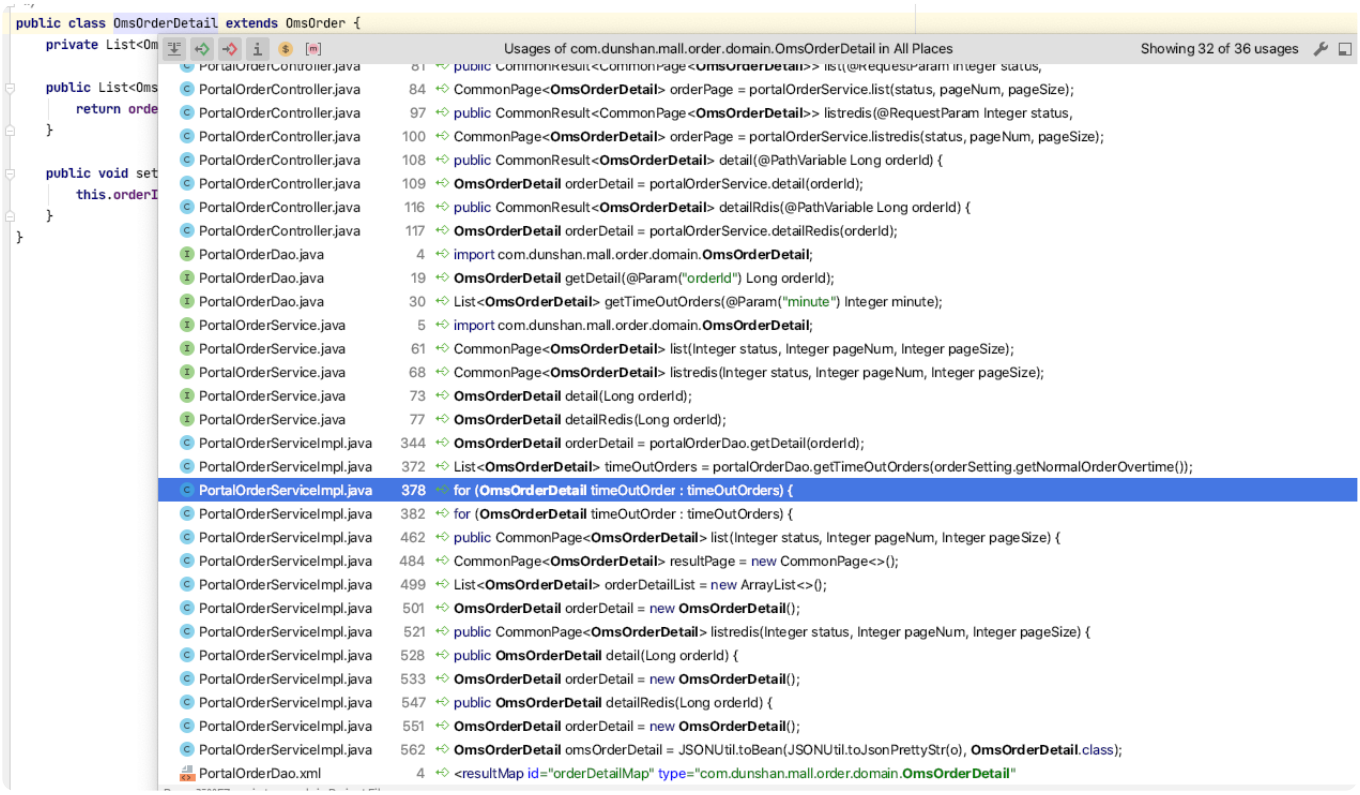
```
<update id="updateByPrimaryKeySelective" parameterType="com.dunshan.mall.model
    update oms_orde
    ......................
    where id = #{id,jdbcType=BIGINT
  </update


<update id="updateByPrimaryKeySelective" parameterType="com.dunshan.mall.model
    update pms_sku_stoc
    ......................
    where id = #{id,jdbcType=BIGINT
  </update
```

```
13    <select id="selectByPrimaryKey" parameterType="java.lang.Long" resultMap="B
14      select
15      <include refid="Base_Column_List" />
16      from pms_sku_stoc
17      where id = #{id,jdbcType=BIGINT
18    </select
```

在我查了这三个 SQL 语句对应的 SQL、表和索引之后，发现它们都是精准查找，并且索引也在第一阶段的分析中创建完了，按理说不会出现大的数据量。可是，我们在前面的确看到 OmsOrderDetail 产生了巨大的数据量，这是怎么一回事？

为了搞清楚这个问题，我们查查还有谁调用了 OmsOrderDetail：



我们找啊找，终于看到了两个 for 循环（其实在前面的代码段中，也可以看出来）。我点进去一看，发现它们是定时任务。这个时候，问题产生的逻辑就变得清晰了：由于存在 for 循环，接口调用时就会循环执行某段代码，这就成了 JVM 内存不断增加的一种可能性。

你还记得吗？在上节课中，我们其实也定位到了“for 循环对应的 SQL 执行慢”这个问题，但是由于压力持续的时间不够长，内存没有被耗尽，所以内存被消耗光的问题并没有体现出来。而当场景执行的时间变长时，就出现了 TPS 断断续续的奇怪现象。

优化效果

到这里，我们的优化方案其实非常清楚了，就是做定时任务时不要一下子查那么多数据。因此我在这里加了一个 limit，限定一次就查 500 行。如果处理不过来的话，我们可以直接写一个单独的服务，进行多线程处理。

修改之后，我们再次看看 Order 服务的 JVM 内存消耗：

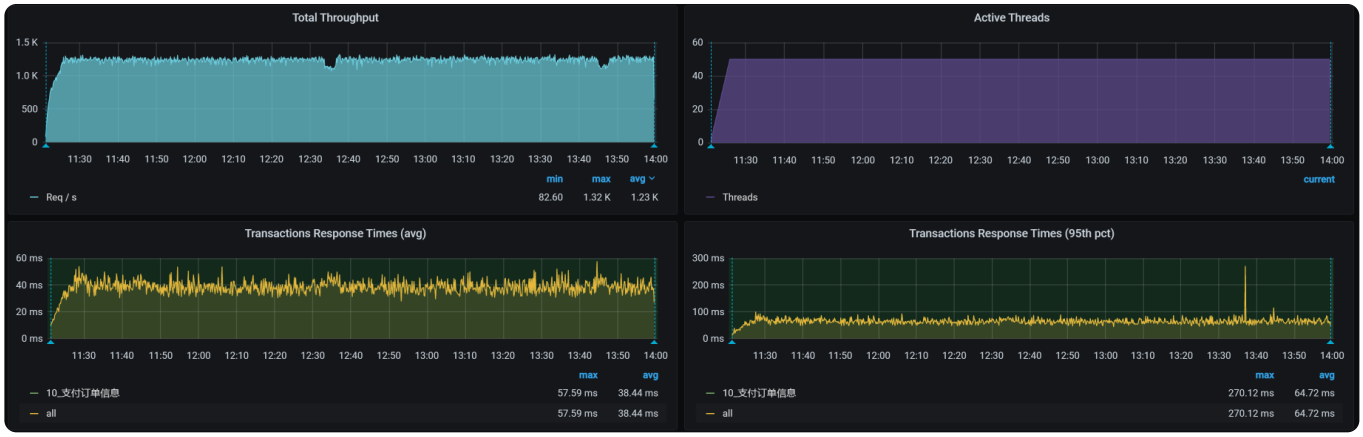
复制代码

```
[root@svc-mall-order-f7b6d6f7-xl2kp /]# jstat -gcutil 1 1s
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   5.44 100.00  96.03  95.03  93.17  43710 1044.757   161   83.635 1128.
  0.00   3.18  82.83  96.21  95.03  93.17  43713 1044.797   161   83.635 1128.
  2.09   0.00   4.54  96.21  95.03  93.17  43718 1044.850   161   83.635 1128.
  1.99   0.00  44.92  96.21  95.03  93.17  43722 1044.891   161   83.635 1128.
  0.00   2.24   1.51  96.22  95.03  93.17  43727 1044.936   161   83.635 1128.
  2.23   0.00   0.00  96.22  95.03  93.17  43732 1044.987   161   83.635 1128.
 40.97   0.00  76.46  96.22  95.03  93.17  43736 1045.051   161   83.635 1128.
  0.00  41.76  47.74  98.81  95.03  93.17  43741 1045.136   161   83.635 1128.
 45.59   0.00  77.61  98.81  95.03  93.17  43746 1045.210   161   83.635 1128.
  0.00   0.00  51.01  52.55  95.03  93.17  43749 1045.270   162   84.021 1129.
 52.34   0.00  60.57  53.23  95.03  93.17  43754 1045.353   162   84.021 1129.
  0.00  51.85   0.00  56.32  95.03  93.17  43759 1045.450   162   84.021 1129.
  0.00  57.97  98.59  58.79  95.03  93.17  43764 1045.526   162   84.021 1129.
 42.02   0.00  83.01  60.83  95.03  93.17  43768 1045.602   162   84.021 1129.
  0.00  42.51  72.69  60.83  95.03  93.17  43773 1045.668   162   84.021 1129.
 28.95   0.00  67.94  61.52  95.03  93.17  43778 1045.735   162   84.021 1129.
  0.00   4.50  32.29  62.74  95.03  93.17  43783 1045.788   162   84.021 1129.
 62.60   0.00  27.04  62.80  95.03  93.17  43788 1045.866   162   84.021 1129.
  0.00  45.52   0.00  65.14  95.03  93.17  43793 1045.950   162   84.021 1129.
  0.00  47.10  71.13  65.14  95.03  93.17  43797 1046.015   162   84.021 1130.
 49.36   0.00  33.30  65.14  95.03  93.17  43802 1046.080   162   84.021 1130.
  5.92   0.00  35.47  67.33  95.03  93.17  43806 1046.132   162   84.021 1130.
  0.00  50.15  65.90  67.37  95.03  93.17  43811 1046.209   162   84.021 1130.
 46.75   0.00  10.39  69.71  95.03  93.17  43816 1046.305   162   84.021 1130.
 47.27   0.00  54.91  69.71  95.03  93.17  43820 1046.364   162   84.021 1130.
  0.00  45.69  46.99  69.71  95.03  93.17  43825 1046.430   162   84.021 1130.
  3.25   0.00  13.65  71.93  95.03  93.17  43830 1046.488   162   84.021 1130.
  0.00  38.00  46.98  71.94  95.03  93.17  43835 1046.551   162   84.021 1130.
 43.74   0.00  37.69  74.44  95.03  93.17  43840 1046.634   162   84.021 1130.
  0.00  42.88  15.64  74.44  95.03  93.17  43845 1046.702   162   84.021 1130.
  0.00  44.13  12.90  74.44  95.03  93.17  43849 1046.756   162   84.021 1130.
  0.00  16.42  33.96  75.79  95.03  93.17  43853 1046.813   162   84.021 1130.
  4.25   0.00  20.10  76.45  95.03  93.17  43858 1046.863   162   84.021 1130.
  0.00   3.22   0.00  76.46  95.03  93.17  43863 1046.914   162   84.021 1130.
```

从 GC 的状态来看，内存现在可以回收得正常一些了。这里我要说明一下，上面的 JVM 数据不是在场景一开始的时候抓取的，而是在场景执行了很久之后才抓取的。因为我们从上面的数据看到 JVM 内存已经可以正常回收了，所以，上面的数据是在确定没有内存溢出的前提下得到的有效数据。

不过，这里还是有问题的，不知道你有没有发现，那就是 YGC 过快。你通过 YGC 这一列就能看到，我一秒打印一次，一秒就有四五次的 YGC，平均每次 YGC 时间大概在 10～20 毫秒。虽然这个问题还没有对 TPS 造成明显的影响，但是也在危险的边缘了。不过，也正因为它现在对 TPS 还没有造成明显的影响，所以，我在这里先不处理 YGC 快的问题了。

我们再来重新执行一遍场景，得到的场景执行结果如下：



你看，TPS 是不是有改善？而且不会再出现第二阶段刚开始的那种情况了，因为当时是压到了 6 分钟之后就开始出现问题了，而在这里，我们压了 30 多分钟，仍然没有出现掉下来的情况。

可以欢呼了，对不对？对的！我们不用再等翻车了。

总结

在这节课中，我们在分两个阶段描述了两个问题。

第一个问题比较简单，是全表扫描没加索引的问题，我们在前面也有过描述。之所以在这节课中又说一下，是因为 SQL 把 CPU 全都吃光了，而这种状态才是全表扫描在性能中比较常见的问题。

第二个问题和内存溢出有关，对于 Java 的应用来说，我们要把内存的溢出找出来，就必须理清楚代码的逻辑，知道哪个变量在哪里定义，谁在取值，谁在使用，还有层级关系也要划分清楚。现在的开发工具已经非常友好了，可以告诉我们代码的前后调用关系。

在这个例子中，难点其实不在于怎么找到这个内存溢出点，而是在于找到溢出的原因。我们只要看下 HeapDump 就可以知道溢出点在哪里，而引用这个类的是谁，可能会有很多地方，我们需要一个个去查看。

这个例子的特殊之处在于，我们的接口本身就使用了 pms_sku_stock 表和 OmsOrderDetail 类，但是从语句上来看，这个接口不会导致内存溢出。所以，我们才需要找出是谁使用了这个 pms_sku_stock 表和 OmsOrderDetail 类，并且会产生溢出。这个转折才是关键。

希望你能在这节课中受益。

课后作业

最后，我给你留两道题，请你思考一下：

1. 你能否快速找到需要创建索引的 SQL？
2. 在内存不断增长时，如何快速定位出哪个对象导致的？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | 支付前查询订单列表：如何分析优化一个固定的技术组件？

下一篇 23 | 决定容量场景成败的关键因素有哪些？

精选留言 (1)

写留言

WBF
2021-05-16

1、mysql：
①查看慢日志
②top -Hp pid 定位线程号，通过线程号定位sql
然后查看执行计划分析是否存在全表扫描，表大小，制定优化方案
oracel：...

展开

作者回复: 整理的不错哦。表扬！
温馨提醒一下，mysql中是没有AWR报告的，oracle里才有。